Formal Verification, Quantitative Analysis and Automated Detection of Design Patterns

Prayasee Pradhan

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India

Formal Verification, Quantitative Analysis and Automated Detection of Design Patterns

Thesis submitted by

Prayasee Pradhan

[Roll: 710CS1029]

In partial fulfilment of the requirements for the award of the degree

of

Master of Technology

in

Computer Science and Engineering

under the guidance of

Prof. S. K. Rath

NIT Rourkela

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India

Department of Computer Science and Engineering
National Institute of Technology Rourkela
Rourkela-769 008, Odisha, India.

May 23, 2015

Certificate

This is to certify that the work in the thesis entitled ***Formal Verification, Quantitative Analysis and Automated Detection of Design Patterns*** by ***Prayasee Pradhan*** is a record of an original research work carried out under my supervision and guidance in partial fulfillment of the requirements for the award of the degree of Master of Technology in Computer Science and Engineering. Neither this thesis nor any part of it has been submitted for any degree or academic award elsewhere.

Prof. S. K. Rath
Professor
Department of Computer Science and Engineering
NIT Rourkela

Acknowledgment

I express my sincere and heartfelt gratitude towards our guide Prof. S. K. Rath for his expert guidance and motivation during the course of the project which served as a spur to keep the work on schedule.

I convey my regards to all the other faculty members of Department of Computer Science and Engineering, NIT Rourkela for their valuable guidance and advices at appropriate times. Finally, I would like to thank Mr. Mohd Suleman and Mr. Ashish Kumar Dwivedi for their help and assistance all through this project.

Prayasee Pradhan

Abstract

Present day software engineering concepts emphasize on developing software based on design patterns. Design patterns form the basis of generic solution to a recurring design problem. The present day software engineering concept emphasizes that software requirement analysis and design methodologies based on different Unified Modeling Language (UML) diagrams need to be strengthened by the use of a number of design patterns. In this study, an attempt has been made for automated verification of the design patterns. A grammar has been developed for verification and recognition of selected design patterns. ANTLR (ANother Tool for Language Recognition) tool has been used for verification of developed grammar.

After proper verification and validation of design patterns, there comes a need to quantitatively determine the quality of design patterns. Hence, we have provided a methodology to compare the quality attributes of a system having design pattern solution with a system having non-pattern solution, both the system intending to provide same functionalities. Using Quality Model for Object-Oriented Design (QMOOD) approach, object oriented metrics are calculated in terms of the number of classes and their relationships in a Unified Modeling Language (UML) class diagram. The cut-off points are calculated in order to provide the exact size of the system in terms of the number of classes, for which the solution adopted using design pattern, provides more quality parameters.

Again Design Pattern Detection (DPD) has also been considered as an emerging fields of Software Reverse Engineering. An attempt has been made to present a noble approach for design pattern detection with the help of Graph Isomorphism and Normalized Cross Correlation(NCC) techniques. Eclipse Plugin i.e., ObjectAid is used to extract Unified Modeling Language (UML) class diagrams as well as the eXtensible Markup Language (XML) files from the Software System and Design Pattern. An algorithm is proposed to extract relevant information from the XML files, and Graph Isomorphism technique is used to find the pattern subgraph. Use of NCC provides the percentage existence of the pattern in the system.

Keywords: Design Patterns; ANTLR; Formal Methods; Object-Oriented Metrics; QMOOD; Quality Attributes, Normalized Cross Correlation, Graph Isomorphism.

Contents

List of Figures

List Of Abbreviation

ANTLR	ANother Tool for Langauge Recognition
AQA	Average Quality Attribute
CASE	Computer Aided Software Engineering
CRG	Correspondence Graph
CVC	Contribution Value of Class
DPD	Design Pattern Detection
DPM	Design Pattern Matrix
DPS	Design Pattern Solution
IDE	Integrated Development Environment
NCC	Normalized Cross Correlation
NPS	Non Pattern Solution
OO	Object Oriented
PG	Pattern Graph
QMOOD	Quality Model of Object Oriented Design
RBAC	Role Based Access Control
SAP	Single Access Point
SG	System Graph
SGM	System Graph Matrix
UML	Unified Modelling Language
XML	eXtensive Markup Language

Chapter 1

Introduction

1.1 Introduction

In the past two decades, a good number of software patterns have been discussed by researchers [1] [2] [3] [4] [5] [6]. Many design pattern tools have also been developed for detecting patterns in instantiating of design patterns [7] [8]. Gamma et al. [1] have proposed the concept of design pattern. They proposed standard templates for tweenty three number of design patterns. Other authors on software design patterns used these templates as a base to further extend or modify these templates for their application areas. Security patterns have been proposed by Yoder and Barcalow [9]. They have proposed seven patterns which are applied in security development issues. After that a good number of other category of security are available in literature[4] [5] [10].

Design of an application system at present is supposed to be based on different UML diagrams. UML class diagram shows the structural behavior of the classes, but it is unable to express some other behavioral aspects. Hence extension of UML diagram to visualize the design pattern methodology was proposed in [11]. It is observed that there is a gradual evolution of representation of design pattern in UML class diagram, incorporating Venn diagram style notation, Dotted-Bounding Pattern Annotation, and Tagged Value Notation. Tagged Value Notation defines the pattern-role behavior of the model elements such as classes, attributes and operations. The verification and validation of any requirement are being carried out using formal languages which are based on grammar and have certain production rules.

According to Yoder [9], secure system should maintain a proper associativity among different security patterns. The first and most important measure for Security Pattern is *Single Access Point* to limit the entry to the System through only a single point. The *Single Access Point* takes the user identification to the *Check Point* for the authentication and authorization of the user. When user identification has been verified, *Session* is created for carrying the global variables defining the user's identification, its role and a connection to a class with the objective of establishing security. The authorization area for system visualization and modification is provided through the Role-Privilege Relationship. Users are provided with the *Limited View* of the whole application or with *Full View of application with Error*.

Presently, object-oriented (OO) paradigm is strongly recommended for software development in contrast to traditional and function-oriented methodologies. Object-oriented methodology has different characteristics, such as encapsulation, polymorphism, and inheritance, which make the code reliable and understandable. Based on these OO principles, different metrics are applied to measure the quality of software using object-oriented methodology [12] [13].

For the quantitative analysis of the quality of software attributes, Bansiya and Davis have proposed a hierarchical model for an object-oriented design quality assessment, called as QMOOD (Quality Model of Object Oriented Design) approach [14]. This model relates the quantifiable object-oriented characteristics to the higher caliber of software quality attributes. QMOOD model is characterized by four levels and three mappings. Figure 1.1 illustrates the structure of QMOOD approach.

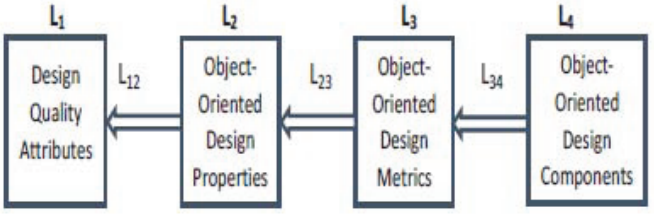

FIGURE 1.1: QMOOD model

First level, L_1, represents the Design Quality Attributes, which are Functionality, Reusability, Flexibility, Understandability, Effectiveness, and Extendibility [14]. The second level, L_2 of QMOOD represents Object-Oriented Design Properties including abstraction, encapsulation, inheritance, polymorphism, coupling, cohesion, and complexity, etc. The third level, L_3 represents the object-oriented design metrics satisfying the design properties. In this work, some of the object oriented design metrics associated with particular design properties are chosen; which are provided in Table 1.1. The fourth level, L_4, represents the Design components, which may be either class or relationship. The first mapping L_{12} between the first level and second level of QMOOD model provides a relation among the various design quality attributes with the design properties, as presented in Table 1.2. The second mapping L_{23} provides the relationship among the design properties and the design metrics. The third mapping L_{34} provides the information about which design metrics are applied to classes or relationship in a particular UML (Unified Modelling Language) class diagram.

TABLE 1.1: Object-Oriented Metric and Description

Sl No	Design Property	Chosen Metric	Metric Name
1.	Design Size	Size	Size of Design
2.	Hierarchies	NOC	Number of Children
3.	Abstraction	DIT	Depth of Inheritance Tree
4.	Encapsulation	DAM	Data Access Metric
5.	Coupling	CBO	Coupling Between Object Classes
6.	Cohesion	CAM	Cohesion Among Methods of a class
7.	Composition	MOA	Message of Aggregation
8.	Inheritance	MFA	Measure of Functional Abstraction
9.	Polymorphism	NOP	Number of Polymorphic Methods
10.	Messaging	RFC	Response for Class
11.	Complexity	WMPC	Weighted Method Per Class

TABLE 1.2: Quality Attributes and Associated Design Properties

Quality Attribute	Design Properties
Reusability	Coupling, Cohesion, Messaging, Design Size
Flexibility	Encapsulation, Coupling, Composition, Polymorphism
Understandability	Abstraction, Encapsulation, Coupling, Cohesion, Polymorphism, Complexity, Design Size
Functionality	Cohesion, Polymorphism, Messaging, Design Size, Hierarchies
Extendebility	Abstraction, Coupling, Inheritance, Polymorphism
Effectiveness	Abstraction, Encapsulation, Composition, Inheritance, Polymorphism

To obtain the abstractions and views from a target system, system developers rely on reverse engineering activities to maintain, evolve and eventually re-engineer the system. Design pattern detection (DPD) is a vast area of research in the field of reverse engineering and reconstruction of software [15] [16]. Detection of design pattern also helps in the re-documentation phase of software development life cycle and enhances the maintainability of the software. Design pattern detection is further useful to provide better comprehension of a software system, its components and its architecture without knowing the details of programming implementations. There are various reasons why it is difficult to detect a design patterns in a software. Finding a design pattern in large software systems is difficult because of larger exploration space. Secondly, a class may play more than one role under different design patterns. Hence, identiification of a design pattern is difficult since it produces ambiguous results. Further, the number of design patterns are increasing day by day. The accuracy of detecting a design pattern can be increased by the application of classification techniques.

Various techniques have been previously adopted to detect design pattern from source code as well as design models [17] [18] [19]. However, these methods are not fully automated.

In this study, a method is proposed by defining the grammar for formal specification of Design Patterns. The system, which follows the proposed grammar, satisfies essential security goals such as integrity, confidentiality, availability, authentication, authorization, and non-repudiation of the desired software. After formal verification of the system, study has been extended towards the quantitative analysis of quality parameters of design patterns. Using QMOOD approach, a software with non-pattern solution can be evaluated to find out the quantitative values for quality attributes and compared with the quality attributes of the existing software using design patterns, where both the software satisfy same functionalities. Cut-off points are provided in terms of the number of classes present in the software for which the design pattern solution provides the best result towards different software quality. This approach may also be helpful to provide a goal-driven software solution. Finally, we have proposed a novel approach to detect design patterns from UML class diagrams automatically. The reverse engineering process of extracting UML class diagram from the source code was done by the help of eclipse plugin i.e, ObjectAid[20]. The extracted xml files corresponding to the system diagram as well as the design pattern diagram are further evaluated

to find the existence of the pattern instances. We have applied our technique on various open source java projects for detection of several design patterns.

1.2 Motivation

During the development of software application, a number of defects grow exponentially with the number of interacting system components. When formalizing the parameters such as, concurrency, non-determinism, and security, it is observed that they are very hard to model using standard designing techniques available in the literature. System's growing size and complexity, together with the pressure of drastically reducing system development time, make the delivery of low-defect systems an enormously challenging and complex activity. Hence, a reusable technique i.e., design pattern is considered to resolve these problems at the very beginning of software development life cycle. But these patterns are semi-formal in nature, hence they need to be verified and validated by using a suitable formal modeling notations. Grammar is one of the formal notations to provide verification and validation technique. Hence, design patterns are verified by the use of developed formal grammar.

Software quality has been recognized as an important topic since the early days of software engineering. Software is being built using traditional methodologies and gradually developers found it simple and reusable to build a system using Object-Oriented (OO) paradigm. Since non-functional requirements are related to object-oriented metrics, it has been proved that maintaining balance among various types of object-oriented metrics enhance a particular non-functional requirement. Hence, the need of quantitative analysis of quality parameters is required. Taking the same system which uses design patten solution as well as non-pattern solution, quantitative analysis of quality attributes provides the solution which gives better quality attribute scores for the system using design pattern than the system using non-pattern solution.

System reverse engineering activities are maintianed to eventually re-engineer the system. Design pattern detection (DPD), being a vast area of research, helps helps in the re-documentation phase of software development life cycle and enhances the maintainability of the software. Several methods are developed to

detect design pattern instances in a system, but most of them are not fully automated. Hence, there is a need to detect design pattern instances in a system. Graph isomorphism technique can be used to detect design pattern in the system to assure the existence of the pattern, and the use of Normalized Cross Correlation technique proves to be a better index of providing the partial existence of pattern. In this study, two approachs to formulate a new method in order to detect design pattern in a system have been combined.

1.3 Organization of the thesis

The thesis is organized as follows.

In chapter 2, the literature survey on different works that has already been undertaken in the field of formal verification of design patterns, quantitative analysis of quality of design patterns, and design pattern detection, has been presented.

In chapter 3, methodology for the formalization of selected design patterns has been discussed. Grammer, which satisfies the proposed pattern language has been further explained.

In chapter 4, a methodology to assess the quality of design patterns has been proposed and experimental details are presented.

In chapter 5, a method to detect design patterns using graph ismorphism and normalized cross correlation techniques has been explained. Also an explained example has been provided to demonstrate our approach.

In chapter 6 presents a conclusion and a focus on future research directions that could be undertaken.

Chapter 2

Literature Survey

2.1 Formal Verification of Design Pattern

It is understood that software testing effort can be decreased by using formal verification techniques. There are several formal verification techniques used so far, which are provided in this section.

The very first notation used for identification of design patterns in UML diagram was *Venn-Diagram style Pattern Annotation* [21]. In this method, the model elements participating under the same pattern are clustered together. The concept is well accepted for small system, but clustering of elements in a larger system was not possible due to the lack of simplicity and overlapping of clusters. This method simply shades the cluster with a color in order to make it distinguishable from other ones, but still it was not widely accepted for large system.

In order to prevent the shortcoming of shading problem of *Venn-Diagram style Pattern Annotation*, the *Dotted-Bounding Pattern Annotation* was developed by Dong [22]. But still the notations were imprecise to decide the exact role of the model elements which they play under the particular design pattern.

Berner et al. have proposed a notation based on UML stereotypes called as *restrictive stereotype* [23]. The method defined the design pattern and role of the model elements participating in a system. But, the stereotype notation was difficult to handle in terms of expensiveness of designing, using and maintaining the notation. Also, their approach was not clear about how to extend UML stereotype notation to represent the compositions of design patterns.

Dong have proposed a new notation to represent explicitly the roles of each class, operation, and attribute in a pattern, which is based on an extension to UML [11]. The extension was defined mainly by applying the UML built-in extensibility mechanisms. The new notation was called as *Tagged Pattern Annotation*. This method also fulfilled the drawbacks of the *Stereotype Annotation Pattern* by allowing the representation of composition of design patterns.

T.Taibi and D.C.L. Neo [24] proposed a formal notation known as, BPSL (Balanced Pattern Specification Language). The main aim of this language was to combine two subsets of Logic, one from First Order Logic (FOL) and other from Temporal Logic of Actions (TLA). According to authors, BPSL has carefully chosen the subsets of FOL and TLA to be used in order to be simple for users

and yet described design patterns accurately. The ultimate purpose of BPSL is to help users to understand patterns to know exactly when and how to use them.

Dong et al. [25] proposed an approach to automate the verification of the compositions of security patterns by model checking. They formally described the behavioral aspect of security patterns in CCS (Calculus of Communicating Systems) through their sequence diagram. They also proved the faithfulness of the transformation from a sequence diagram to its CCS representation. In their research, they used two case studies to demonstrate their approach and shown its capability to detect composition errors. Dwivedi and Rath [26] formalized a complex architectural style i.e., C2 (Component and Connector) using formal modeling language Alloy. They have considered cruise control system as a case study.

Bayley and Zhu [27] proposed a meta-modeling approach toward formalization of design patterns. This approach enables formal reasoning about patterns and their composition, transformation, and facilitates automatic tool support for applying patterns at the design stage. For the case study, authors have formally specified all 23 Gamma's design patterns. They claimed that the class diagram of facade pattern given by GoF [1] is not even well-formed and cannot be taken at face-value in terms of either the number of classes or their inter-connections. Dwivedi and Rath [28] have formalized an architectural style C2 using formal modeling languages Alloy and Promela. For the model checking of these formal notations, automated verifiers such as Alloy Analyzer and SPIN are used.

Dey and Bhattacharya [29] have proposed a formal specification language FSDP (Formal Specification of Design Pattern) to formally specify design patterns from UML class diagram. They have used ANTLR (ANother Tool for Language Recognition) for verification of their developed grammar. They developed a tool from FSDP grammar to formally automate pattern design techniques, to create, store, and retrieve UML class diagrams within design patterns. The proposed grammar is only able to verify the notation [11] for representing design patterns in extended UML class diagram. Grammar verifies textual format of extended UML class diagram but it does not check associativity between the different design patterns and it also fails to check correct placement of roles for design patterns.

2.2 Quantitative Analysis of Quality Parameters

The amount of work that has been undertaken till date on the field of quantitative analysis for quality of design patterns are provided in this section.

Ampatzoglou et al. presented a methodology to compare the design pattern with alternative solutions considering several quality attributes [30]. They proposed a methodology to find out major changes of axes in the design patterns and provided metric scores in terms of the number of classes, which form the major axes of change. The authors mentioned about the cut-off points for different metric scores for the Bridge design pattern. But they did not mention about the cut-off points for other quality attributes. Issaoui et al. presented a metric-based filtering approach to improve software design patterns detection technique [31]. They have shown a number of case studies, such as JHotDraw v5.1, JRefactory v1.0, QuickUML2001, etc. for the evaluation of metric values for GoF design patterns.

Chang et al. presented the benefits of design pattern based framework [32]. They performed a quantitative analysis on pattern-based system to check the improvement of quality parameters, such as abstraction, usability, complexity etc. Hsueh et al. adopted a quantitative approach for evaluating the quality of design patterns [33]. Authors suggested a method to validate whether a design pattern is well-designed or not. But the drawback of the work is that the method can be applied on a single design pattern taking a single object-oriented metric into consideration. This work does not consider the effect of the design patterns on the other design metrics. Hence, taking conclusion that the use of a particular design pattern should be adopted because it enhances a single design metric, is not safe.

Ampatzoglou et al. presented a mapping result of a number of papers, which are based on GoF design patterns [34]. They described the effect of software design patterns (GoF design patterns) quality parameters, such as metrics, usability, complexity, maintainability, adaptability, reliability, etc. Dong et al. proposed a design pattern visualization approach [35]. For the demonstration of their technique, they performed quantitative evaluation of object-oriented attributes. Kataoka et al. proposed a quantitative approach to measure the maintainability of program refactoring [36]. They considered the coupling metrics to evaluate degree of maintainability enhancement. They applied their approach to different programs for the comparative study. Brain Huston suggested a method for improving software

quality by collecting metric scores for a given design [37]. But this work does not consider the effect of the design pattern on other metric scores.

2.3 Design Pattern Detection

Albin-Amiot et al. proposed an approach to use a meta-model in order to obtain a representation of design patterns which will further allow both automatic code generation and design pattern detection [38]. Heuzeroth et al. proposed a method to detect design patterns in legacy code combining static and dynamic analyses [39]. They have developed a tool and classified potential pattern instances according to the information provided by their tool. They have provided their analyses for various design patterns on the Java SwingSetExample.

Begenti e al. have presented a system called IDEA (Interactive Design Assistant), which can automatically detect patterns in a UML class diagram and can also produce critiques about the detected patterns [40]. They have also integrated the concept of IDEA with the CASE tool Argo/UML. Gupta et al. have applied a graph matching algorithm to detect design patterns in the UML class diagram of a system [41]. The algorithm decomposes the graph matching process into K phases, where K ranges from 1 to the minimum number of the numbers of nodes in the two graphs to be matched.

Wenzel et al. have proposed a method to detect design pattern instances in software systems regarding model-driven development [42]. Their proposed approach allows developer to use UML diagram editors to specify patterns. They have used a difference algorithm called as SiDiff to compute the differences between graph-structured UML diagrams. Antoniol et al. have proposed an approach based on multi-stage reduction strategy using object-oriented (OO) software metrics and structural properties to extract structural design patterns from OO design [43]. They have also developed a tool to assess the effectiveness of the approach.

Gupta et al. have provided an approach to detect the design patterns by the application of normalized cross correlation while taking design pattern as a template to find its presence in the system design [44]. Ba-Brahem et al. have proposed an approach to detect design pattern instances in a system design which uses the graph implementation to produce both the system as well as the design pattern UML diagrams in Graph of 4-tuples elements [45].

Tsantails et. al. have proposed a methodology to detect a design pattern based on similarity scoring between graph vertices which is capable of recognizing patterns that are modified from their standard representations [46]. Instead of relying on pattern-specific heuristic, the approach reduces the search space by taking the fact into consideration that pattern resides in one or more inheritance hierarchies.

Dong et al. have adopted a template matching algorithm to detect design patterns from a software system by the use of normalized cross correlation [47]. They have extracted exact matches as well as partial instances for design patterns. Dongjin et al. have proposed a method to detect deign pattern instances in which they have identified all the candidate classes in the system graph satisfying pattern classes [48]. They have selected some of candidate classes to form the sub-graphs to check their isomorphic behavior towards the pattern graph corresponding to the design pattern.

Chapter 3

Formalization of Design Patterns

3.1 Introduction

Design patterns are the generic solution to the mostly recurring problems. Design patterns form a specific association among themselves with in a system in order to perform specific functions. The four design patterns are *Single Access Point*, *Check Point*, *Role*, and *Session*. The pattren language for these design patterns are shown in Figure 3.1, Figure 3.2, Figure 3.3, and Figure 3.4 respectively.

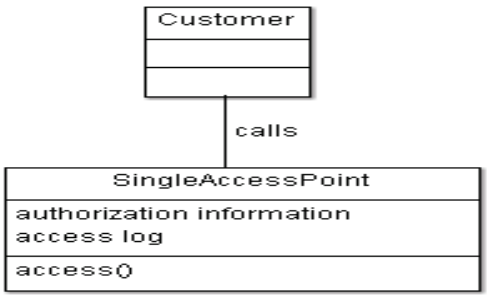

FIGURE 3.1: Pattern Language for Single Access Point

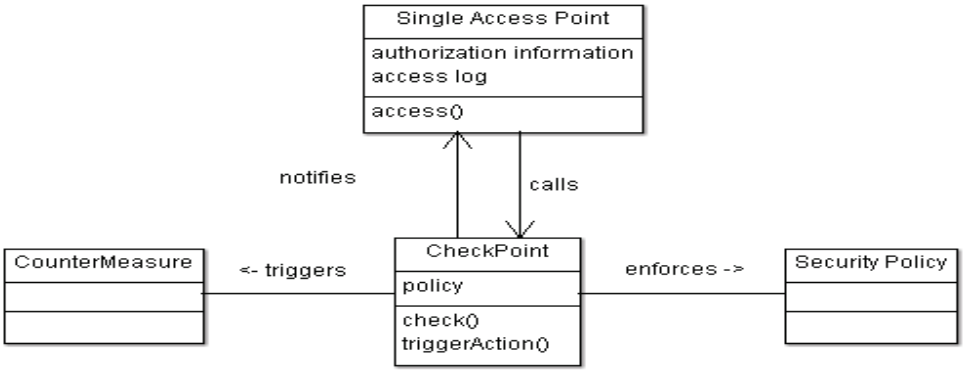

FIGURE 3.2: Pattern Language for CheckPoint

According to Yoder and Barcalow [9], secure system should maintain a proper associativity among different design patterns. *Single Access Point* limits the entry to the system only through a single entry point and provides user identification related information to *Check Point* for authentication and authorization of the user. When user identification has been verified, *Session* is created for carrying the global variables which contain user's identification and its role. The authorization area for system visualization and modification is provided through role-privilege relationship.

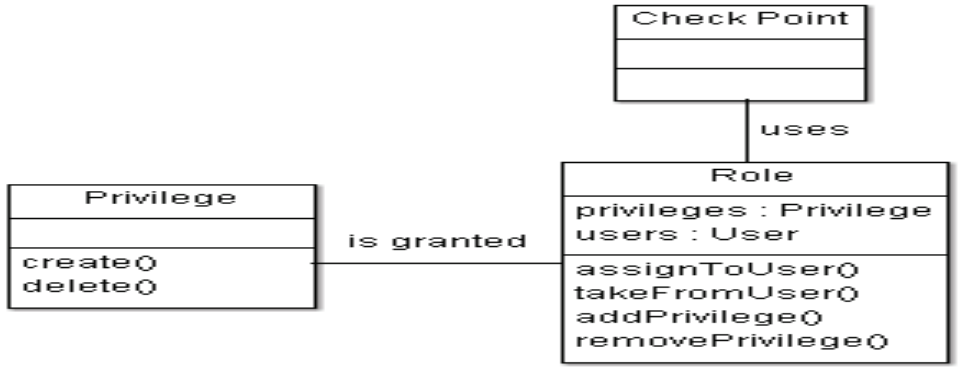

FIGURE 3.3: Pattern Language for Role

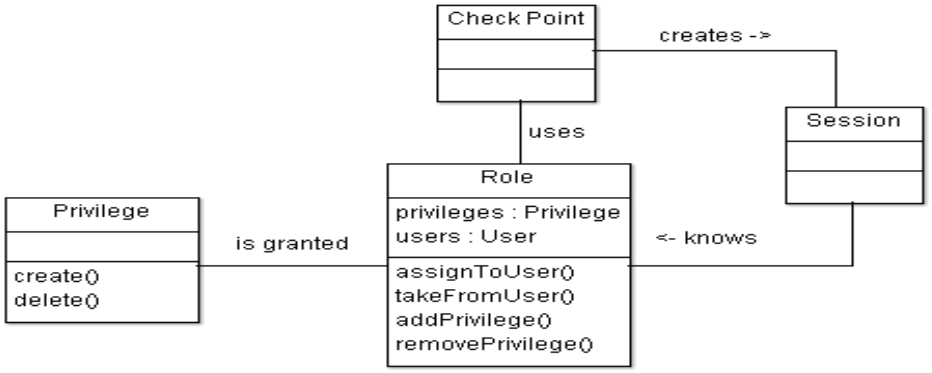

FIGURE 3.4: Pattern Language for Session

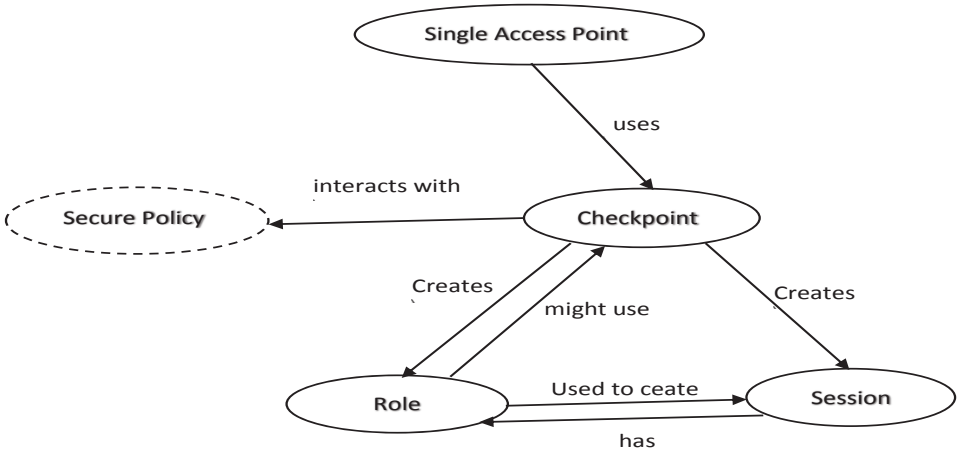

FIGURE 3.5: Pattern Language for selected patterns

3.2 Proposed Approach

A formal specification in the form of grammar is proposed for verification and validation of design patterns. Proposed grammar is based on the pattern language which is shown in Figure 3.5. Four design patterns taken into consideration are *Single Access Point*, *Check Point*, *Session*, and *Role*. The system may contain other design patterns, but presence of design patterns in a particular association is of very much significance for secure system. Any language which is accepted by the proposed grammar, may be said to preserve security aspects. With the help of proposed tool, user can add design pattern in UML class diagram in order to generate the extended UML class diagram. The extended UML diagram is verified with the help of proposed grammar.

The main aim of this paper is to verify the associativity of Pattern-Class containing the aforesaid design patterns. The grammar is developed according to the specification of ANTLR. The parser rule and lexer rules for the proposed grammar are given in Figure 3.6 and Figure 3.7 respectively.

3.2.1 Illustrative Example

In order to demonstrate our approach, online banking system have been considered as a case study. Nowadays, customers need more advocacy, more personal security and, more control in their banking relationships. The major challenge with different banks is that they are looking to gain the flexibility, shared services, easy to use and align business to technology. The solution of above challenges can be found with the help of design patterns. In online banking system, customer performs online financial transactions, which requires more security provision. The UML class diagram and extended UML class diagram for the online banking system are presented in Figure 3.8 and Figure 3.9 respectively.

Figure 3.8 shows the class diagram of Online Banking System for incorporating security features. This diagram contains eleven classes such as *Customer*, *Login*, *Verification*, *SecurePolicies*, *Penalities*, *Sessions*, *ManagingRoles*, *UserPrivileges*, *AccountManagement*, *TransferFund*, and *BalanceEnquiry*. Figure 3.9 is extended for the visualization of design patterns. Extended UML class diagram along with the visualization of design patterns is represented in Figure 3.9. Explanation

```
classDecl : className sapExternal;
sapExternal : LEFTBRACE ( patternSAP (patternInstance)? SLASH roleSAPExternal (COMMA)?)+ ((COMMA)? patternName
(patternInstance)? SLASH role)* RIGHTBRACE classSAPSingleton;
classSAPSingleton : LEFTBRACKET className sapSingleton;
sapSingleton : LEFTBRACE ( patternSAP (patternInstance)? SLASH roleSAPSingleton (COMMA)?)+ ((COMMA)? patternName
(patternInstance)? SLASH role)* RIGHTBRACE (RIGHTBRACKET)? classCheckPoint;
classCheckPoint : LEFTBRACKET className checkPoint;
checkPoint : LEFTBRACE ( patternCheckPoint (patternInstance)? SLASH roleCPCheckPoint (COMMA)?)+ ((COMMA)?
patternSAP (patternInstance)? SLASH roleSAPInternal)+ ((COMMA)? patternName (patternInstance)? SLASH role)*
RIGHTBRACE (RIGHTBRACKET)? classSecurityPolicy;
classSecurityPolicy : LEFTBRACKET className securityPolicy;
securityPolicy : LEFTBRACE ( patternCheckPoint (patternInstance)? SLASH roleSP (COMMA)?)+ ((COMMA)? patternName
(patternInstance)? SLASH role)* RIGHTBRACE (RIGHTBRACKET)? classCounterMeasure;
classCounterMeasure : className counterMeasure;
counterMeasure : LEFTBRACE ( patternCheckPoint (patternInstance)? SLASH roleCM (COMMA)?)+ ((COMMA)? patternName
(patternInstance)? SLASH role)* RIGHTBRACE (RIGHTBRACKET)? classSession;
classSession : className session;
session : LEFTBRACE ( patternSession (patternInstance)? SLASH roleSession (COMMA)?)+ ((COMMA)? patternName
(patternInstance)? SLASH role)* RIGHTBRACE (RIGHTBRACKET)? classRole ;
classRole : className rbac;
rbac : LEFTBRACE ( patternRBAC (patternInstance)? SLASH roleROLE (COMMA)?)+ ((COMMA)? patternName
(patternInstance)? SLASH role)* RIGHTBRACE (RIGHTBRACKET)? classPrivilege ;
classPrivilege : LEFTBRACKET className rolepg;
rolepg : LEFTBRACE ( patternRBAC (patternInstance)? SLASH rolePrg (COMMA)?)+ ((COMMA)? patternName
(patternInstance)? SLASH role)* RIGHTBRACE (RIGHTBRACKET)? finalsessionclass ;
finalsessionclass : className finalsession;
finalsession : LEFTBRACE ( patternSession (patternInstance)? SLASH roleSession (COMMA)?)+ ((COMMA)? patternName
(patternInstance)? SLASH role)* RIGHTBRACE (RIGHTBRACKET)? endclass;
endclass : (RIGHTBRACKET)+ ((LEFTBRACKET)? (className patternDecl)+ (RIGHTBRACKET)* )*;
patternDecl : LEFTBRACE patternName (patternInstance)? SLASH role (COMMA patternName (patternInstance)? SLASH role)*
RIGHTBRACE;
patternInstance : LEFTBRACKET instanceNo RIGHTBRACKET;
patternSAP : SINGLEACCESSPOINT;
roleSAPExternal : EXTENTITY;
roleSAPInternal: INTENTITY;
roleSAPSingleton : SAPSINGLETON;
patternCheckPoint : CP;
roleCPCheckPoint : CP;
roleSP : SP;
roleCM : CM;
patternSession : SS;
roleSession : SS;
patternRBAC : 'RABC';
roleROLE : 'Role';
rolePrg : 'Privilege';
entitynumber : INTEGER;
instanceNo : INTEGER;
className : STRING;
patternName : STRING;
role : STRING;
```

FIGURE 3.6: Parser Rule

```
EXTENTITY : 'ExternalEntity';
SAPSINGLETON : 'Singleton';
INTENTITY : 'InternalEntity';
CP : 'CheckPoint';
SP : 'SecurityPolicy';
CM : 'CounterMeasure';
SS : 'Session';
WS : [ \t\r\n]+ ->skip;
COMMA : ',';
SLASH: '/';
LEFTBRACE : '{';
RIGHTBRACE : '}';
LEFTBRACKET : '[';
RIGHTBRACKET : ']';
SINGLEACCESSPOINT : 'SAP';
STRING : (CHAR)+;
INTEGER : (DIGIT)+;
CHAR : ('a'..'z'|'A'..'Z')('a'..'z'|'A'..'Z' | '|' | '_'| '.'| '-'| '0'..'9')*;
DIGIT : ('0'..'9');
```

FIGURE 3.7: Lexer Rule

of design patterns as visualized in extended UML class diagrams and how these design patterns help in achieving the security goals is explained in the following paragraphs.

For an online banking system, customer is the external entity to interact with the system.To provide clearly defined entrance to all the external entities SAP(Single Access Point) design pattern is considered.*Customer* class plays the role of *ExternalEntity* which is a participant of *SingleAccessPoint* design pattern. Therefore, stereotype notation for *Customer* class is *Customer* {*SAP / ExternalEntity*} which is represented as 'CLASSNAME {PATTERN NAME/ROLE NAME}'. Customer opens the login screen to enter the system which is the only entry point to the system. Accordingly, stereotype notation for *Login* class is {*SAP / Singleton*}.

Customer authenticates itself by providing his required authentication information, this information is used for the verification of customer identity. *Verification* class verifies this information and authenticates the user depending on the security policies enforced by the system. *CheckPoint* is used for implementing security policies as required by the system and it is also used for penalizing the user for violating security policies. Verification class also plays the role of *InternalEntity* under the design pattern *SingleAccessPoint*. After the addition of roles, stereotype annotation for *Verification* class becomes *Verification* {*SingleAccessPoint/*

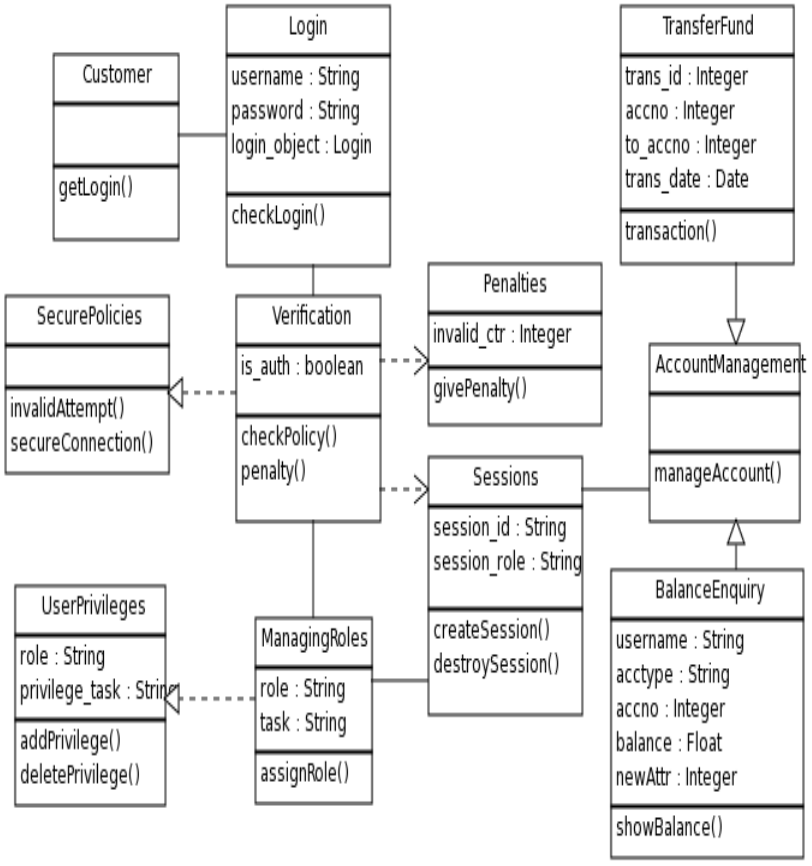

FIGURE 3.8: UML Class Diagram for Online Banking System

InternalEntity, CheckPoint/CheckPoint}.

User authentication is checked in *Verification* class and if the user is not identified, then method of *Verification* class triggers an action to impose penalty. The *Penalties* class performs a role of *CounterMeasure* under the pattern *CheckPoint*. Stereotype annotation of *Penalties* class becomes {*CheckPoint/CounterMeasure*}. After the authentication of user, system needs to identity the authorized area and restricted area for identified user, for this purpose RABC (Role Based Access Control) design pattern is used. When user is authenticated, its role is retrieved from the class *ManagingRoles* which plays the role of *Role* under the *Role Based Access Control* design pattern and its authorized area is retrieved from the class *UserPrivileges* which plays the role of *Privilege* user the design pattern *RBAC*. Class which plays the role of *Privilege* must be associated with the class which plays the role of *Role* under the design pattern *RBAC*. These associations are checked by proposed tools, which is discussed in next section.

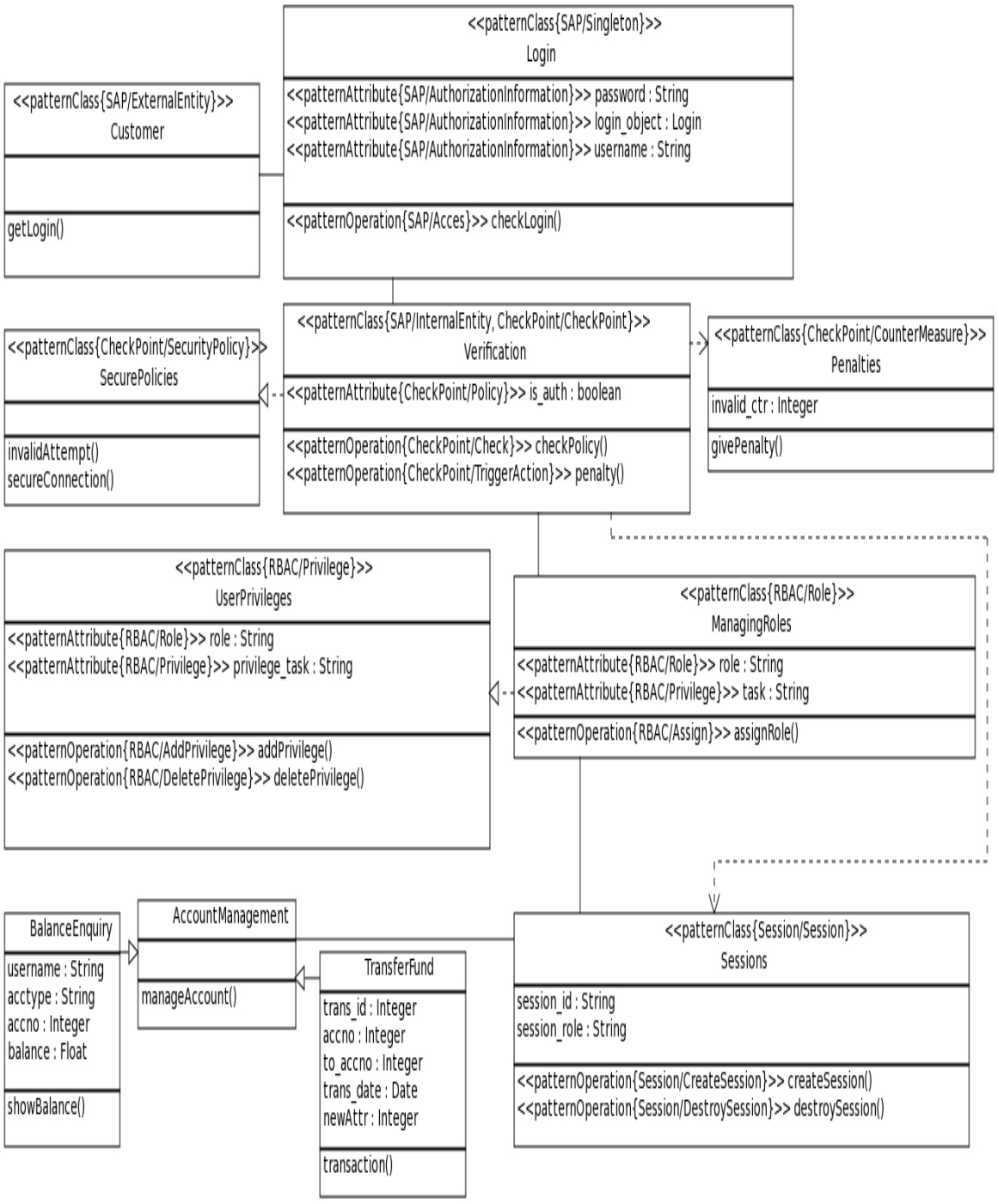

FIGURE 3.9: Extended UML class Diagram for Online Banking System

After the verification and recognition of the role and privileges of user, session must be created to store the global variables in order to keep track of the user identification information such identity, role and privilege. All other classes developed for handling actions such as transfer, withdrawal , deposit must be attached to session class, because session contains the global variables which hold information about the role and privileges of user. *Session* design pattern has been used for creating session and for storing global variables in order to secure the restricted areas. *Sessions* class performs the role of *Session* under the *Session* design pattern. All the other classes such as *BalanceEnquiry*, *AccountManagement*, *TransferFund* play the role of system component which uses sessions.

The above details show, how the four selected design patterns are helpful in achieving desired security goals. Every system which aims at providing a single entry point, user authentication, role and privileges for user, and needs to maintain session, can be made secure at the time of system design by applying four selected design patterns according to the pattern language shown in Figure 3.5.

3.2.2 Test Cases

In order to explain the verification process which is performed by proposed tool, two test cases have been considered as shown in the Figure 3.10 and Figure 3.11. These test cases are the class associativity files generated by tool from the class text of extended UML class diagram.

First test case i.e., Figure 3.9 is generated from the extended UML class diagram which is shown in Figure 3.8. This test case is accepted by the proposed tool because it strictly follows the pattern language as shown in Figure 3.5.

Second test case i.e., Figure 3.10 satisfies all the production rules according to the proposed grammar which is developed for pattern language shown in Figure 3.5. Therefore, it is accepted by the proposed tool. Difference between the first and second test case is as follows: First test case does not contain any design pattern other than the four selected design patterns, for which the pattern language is composed. Second test case contains four selected design patterns as well as other Gamma et.al. design patterns but at the same time it is in accordance with the pattern language shown in Figure 3.5.

```
Customer{SAP/ExternalEntity}
 [Login{SAP/Singleton}
  [Verification{CheckPoint/CheckPoint, SAP/InternalEntity}
   [SecurePolicies{CheckPoint/SecurityPolicy}
    Penalties{CheckPoint/CounterMeasure}
    Sessions{Session/Session}
    ManagingRoles{RABC/Role}
    [UserPrivileges{RABC/Privilege}
     Sessions{Session/Session}
     [AccountManagement{NULL/NULL}
     [BalanceEnquiry{NULL/NULL}
      TransferFund{NULL/NULL}
     ]
     ]
   ]
  ]
 ]
]
```

FIGURE 3.10: Test-Case 1

```
Customer { SAP/ExternalEntity, AbstractFactory/AbstractProduct}
 [Login { SAP/Singleton, AbstractFactory/createProduct}
  [ Verification{CheckPoint/CheckPoint, SAP/InternalEntity, Adapter/Adapter}
   [ SecurePolicies{CheckPoint/SecurityPolicy}
    Penalties{CheckPoint/CounterMeasure}
    Sesssions{Session/Session, Factory/Creator}
    ManagingRoles{RABC/Role}
    [ UserPrivileges{RABC/Privilege}
     Sessions{Session/Session}
     [
      AccountManagement{AbstractFactory/AbstractFactory}
      [
       BalanceEnquiry{AbstractFactory/createProduct}
       TransferFund{ABstractFactory/createProduct}
      ]
     ]
    ]
   ]
  ]
 ]
]
```

FIGURE 3.11: Test-Case 2

Chapter 4

Quantitative Analysis of Quality of Design Patterns

4.1 Introduction

Quality Model of Object-Oriented Design(QMOOD) is the basis of quantitative analysis of quality parameters of design patterns as QMOOD approach relates the quantifiable object-oriented characteristics to the higher caliber of software quality attributes.Using QMOOD approach, a software with non-pattern solution is evaluated to find out the quantitative values for quality attributes and compared with the quality attributes of the existing software using design patterns, where both the software satisfy same functionalities.

4.2 Proposed Work

A method has been proposed to compare the software solutions with use of design patterns and without use of design patterns taking a number of object-oriented metrics into consideration. Case studies for Bridge, Visitor and Abstract Factory design pattern are adopted in order to prove the methodology. The QMOOD approach [14] has been adopted to evaluate the quality attributes of the software system before applying design pattern and after applying the design pattern. The proposed methodology is as follows:

1. Total number of classes in the UML class diagram of the system before using design pattern and after using design pattern, are identified.
2. Based on the definition, various metrics for the two structures of the same system, i.e. before using design pattern and after using design pattern, are found out.
3. Average metric scores are found by dividing the metric scores by the total number of classes.
4. The average quality attribute score using average metric scores for the non-pattern solution and the design pattern solution of the system are found out.
5. For each quality attribute, the quality score difference, i.e. the difference between the average quality attribute score of the system having non-pattern solution and that of the system having design pattern solution are found out.

6. To find the cut-off point for which design pattern solution promotes better result of quality attribute values, the following in-equality is solved:

$$(AQA)_{DPS} - (AQA)_{NPS} \geq 0 \tag{4.1}$$

Where $(AQA)_{DPS}$ = Average Quality Attribute Score of the system after applying design pattern and
$(AQA)_{NPS}$ = Average Quality Attribute Score of the system before applying design pattern.

4.2.1 Illustrative Example

This approach is being illustrated using Visitor design pattern in a system. Visitor design pattern, a behavioral design pattern[1], is used in a scenario when an operation is needed to be performed on elements of an object structure. Visitor design pattern helps in defining a new operation without changing the classes of the elements on which it operates. The UML class diagram for Visitor Non-Pattern Solution and Visitor Design Pattern solution are shown in Figure 4.1 and Figure 4.2 respectively [1, 37].

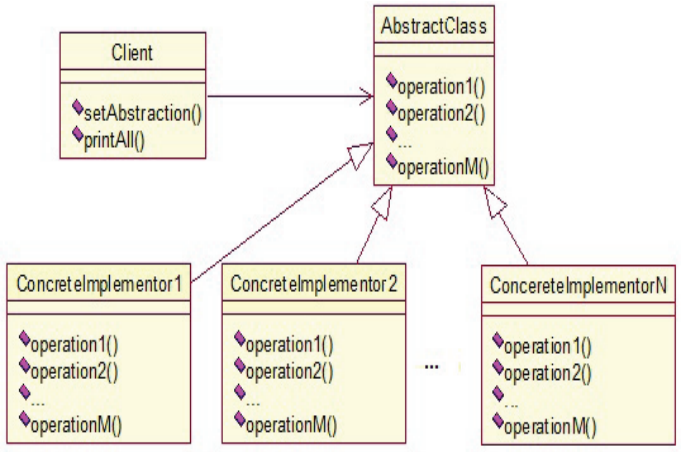

FIGURE 4.1: Class Diagram for Visitor Non-Pattern Solution

4.2.1.1 Average Metric Score Calculation for System having Visitor Non-Pattern Solution

- Let, Total number of ConcreteImplementor class = n;

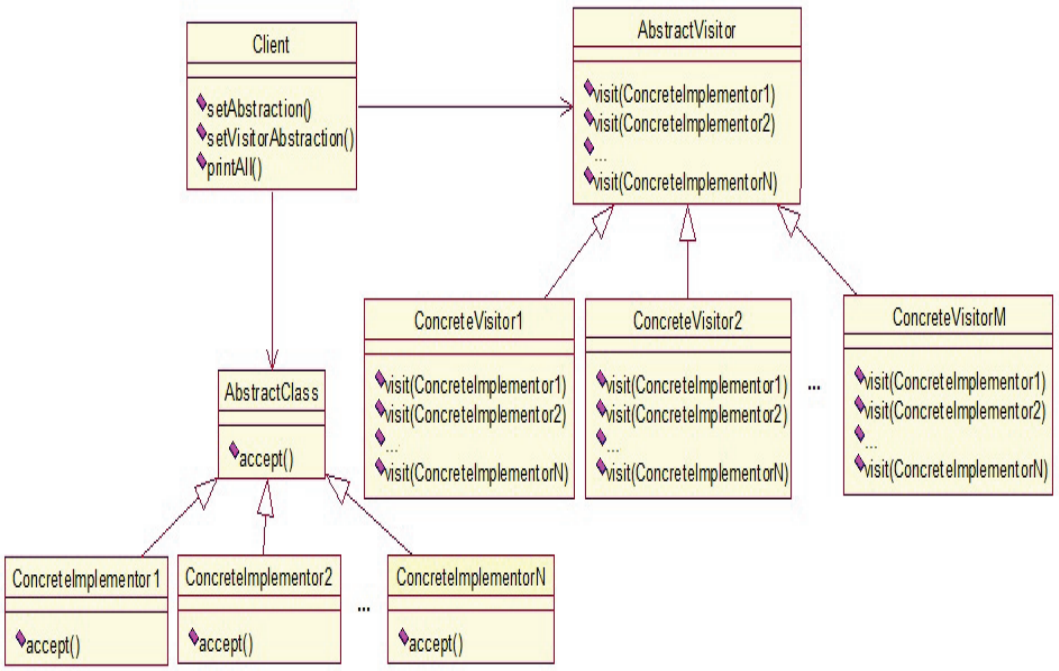

FIGURE 4.2: Class Diagram for Visitor Pattern Solution

- Total number of operations to be performed = m;
- Total number of classes = n+2 (as shown in Figure 4.1)

The average metric scores in terms of number of classes are calculated for the system without using Visitor design pattern. The values are listed in Table 4.1.

4.2.1.2 Average Metric Score Calculation for System having Visitor Design Pattern Solution:

- Let, Total number of ConcreteImplementor class = n;
- Total number of operations to be performed = m;
- Total number of ConcreteVisitor classes = m
- Total number of classes = n+m+3 (as shown in Figure 4.2)

The average metric scores in terms of number of classes are calculated and listed in Table 4.2.

TABLE 4.1: Average Metric Score for Visitor Non-pattern Solution

Sl No	Metric	Average Metric Value	Explanation
1.	*SIZE*	$\frac{nm+m+2}{n+2}$	No. of methods in Client class is 2. No. of methods in AbstractClass is 'm'. No. of methods in ConcreteImplementor class is 'm'.
2.	*NOC*	$\frac{n}{n+2}$	No. of children for AbstractClass is 'n', for other classes, it is 0.
3.	*DIT*	$\frac{n}{n+2}$	For ConcreteImplementor class, DIT is 1, for other classes, it is 0.
4.	*DAM*	No Meaning	No attribute is taken into consideration.
5.	*CBO*	$\frac{1}{n+2}$	No. of classes associated with Client class is 1, for others, it is 0.
6.	*CAM*	No Meaning	No operations other than polymorphic operations are considered in the UML class diagram, hence CAM metric can not be determined.
7.	*MOA*	0	No part-whole relation exists in the UML class diagram.
8.	*MFA*	No Meaning	It is the ratio of total number of inherited methods to the total number of methods accessed by the class. In the case study, only polymorphic methods are taken into consideration.
9.	*NOP*	$\frac{m}{n+2}$	No. of Polymorphic Methods in AbstractClass is 'm', for other classes, it is 0.
10.	*RFC*	$\frac{2+2m+nm}{n+2}$	Client class has 2 local methods and it can invoke 'm' number of methods in AbstractClass class. For Client class, RFC is 2+m. AbstractClass has 'm' number of methods and each of ConcreteImplementors class has 'm' number of methods.
11.	*WMPC*	$\frac{2+m+nm}{n+2}$	For the Client class, Cyclomatic Complexity = 2. For the Abstraction class, Cyclomatic Complexity is 'm'. For the 'n' classes that represent ConcreteImplementors, Cyclomatic Complexity is 'm'.

4.2.1.3 Quality Attribute Score Computation and Finding Cut-off Points

According to Table 1.2, quality attribute scores are calculated which takes the metrics values into consideration. Average Quality Attribute Scores for non-pattern solution and design pattern solution are calculated and difference was found out. In order to find the cut-off points for which design pattern solution gives better result of quality attribute score, The inequality in equation 4.1 has been solved.

TABLE 4.2: Average Metric Score for Visitor Design Pattern Solution

Sl No	Metric	Average Metric Value	Explanation
1.	*SIZE*	$\frac{2n+nm+4}{n+m+2}$	No. of methods in Client class is 3. No. of methods in AbstractClass is 1. No. of methods in ConcreteImplementor class is 1. AbstractVisitor and ConcreteVisitor classes contain 'n' number of operations.
2.	*NOC*	$\frac{n+m}{n+m+2}$	No. of children for AbstractClass is 'n', for AbstractVisitor class, it is 'm', and for other classes, it is 0.
3.	*DIT*	$\frac{n+m}{n+m+2}$	For ConcreteImplementor class, DIT is 1,for ConcreteVisitor class, DIT is 1, and for other classes, it is 0.
4.	*DAM*	No Meaning	no attribute is taken into consideration.
5.	*CBO*	$\frac{2}{n+m+2}$	No. of classes associated with Client class is 2, for others, it is 0.
6.	*CAM*	No Meaning	No operations other than polymorphic operations are considered in the UML class diagram, hence CAM metric can not be determined.
7.	*MOA*	0	No part-whole relation exists in the UML class diagram.
8.	*MFA*	No Meaning	It is the ratio of total number of inherited methods to the total number of methods accessed by the class. In the case study, only polymorphic methods are taken into consideration.
9.	*NOP*	$\frac{n+1}{n+m+2}$	No. of Polymorphic Methods in AbstractVisitor class is 'm', for AbstractClass, it is 1, and for other classes, it is 0.
10.	*RFC*	$\frac{5+3n+nm}{n+m+2}$	Client class has 3 local methods and it can invoke 'n' number of methods in AbstractVisitor class as well as the single method in AbstractClass. For Client class, RFC is 4+n. AbstractVisitor and ConcreteImplementor classes have 'n' number of methods. AbstractClass and each of ConcreteImplementor classes have got one(1) method.
11.	*WMPC*	$\frac{2n+nm+4}{n+m+2}$	For the Client class, Cyclometic Complexity is 3. For the Abstraction class, Cyclometic Complexity is 1. For ConcreteImplementor class, Cyclometic Complexity is 1. For AbstractVisitor class, it is 'n'. For ConcreteVisitor class, it is 'n'.

Let Quality Attribute be "Reusability" [14]

Reusability = -0.25*Coupling + 0.25*Cohesion + 0.3* Messaging + 0.5 * Design Size

Relating the design properties with metrics; Reusability Score can be calculated

as

Reusability Score = -0.25* CBO + 0.25* CAM + 0.5*RFC +0.5*SIZE

Average Reusability Score of Non-Pattern Solution = $(Reusability)_{NPS}$

Average Reusability Score of Design Pattern Solution = $(Reusability)_{DPS}$

Average Reusability Score Difference = $(Reusability)_{DPS}$ - $(Reusability)_{NPS}$

To find the cut-off points, the inequality to be solved is

$$(Reusability)_{DPS} - (Reusability)_{NPS} \geq 0 \tag{4.2}$$

The above inequality is satisfied for

$$m >= 1;\ n >= \frac{\sqrt{(32m^2+144m+137)}-11}{8} \tag{4.3}$$

Similarly other quality attribute scores are calculated and cut-off points are estimated by solving the general equation 4.1.

TABLE 4.3: Quality Attribute Cut Off points for Design patterns

Sl No.	Quality Attribute	Visitor	Bridge	Abstract Factory
1.	Resuability	$m >= 1;\ n >= \frac{\sqrt{(32m^2+144m+137)}-11}{8}$	$m >= 1;\ n >= \frac{\sqrt{(4m^6-28m^5+41m^4+364m^3+402m^2-304m+41)}}{2(5m-2)} + \frac{2m^3-7m^2-12m+3}{2(5m-2)}$	$m >= 1;\ n >= 3$
2.	Flexibility	$m >= 1;\ n >= \frac{\sqrt{(20m^2+20m+1)}+(2m-5)}{4}$	$m > 2;\ n >= \frac{\sqrt{(m^4-6m^3+19m^2-14m+9)}}{2(m-2)} + \frac{m^2-3m+7}{2(m-2)}$	$m < 3;\ n >= 1$
3.	Understandability	$m >= 1;\ n >= 1$	$m >= 1;\ n >= \frac{\sqrt{(4m^6-32m^4+44m^3+340m^2+412m+121)}}{2(2m+3)} + \frac{2m^3-12m-13}{2(2m+3)}$	No Integer Solution
4.	Functionality	$m < 3;\ n >= 1$	$m > 0;\ n >= \frac{\sqrt{(121m^6-506m^5+3675m^4+16704m^3+65407m^2+18786m-407)}}{2(45m+22)} + \frac{11m^3-23m^2+53m-33}{2(45m+22)}$	$m >= 1;\ n >= 2$
5.	Extendability	$m >= 1;\ n >= \frac{\sqrt{(5m^2-2m-3)}+(m-1)}{2}$	$m >= 1;\ n >= \frac{\sqrt{(9m^4+58m^3+129m^2+96m+32)}}{2(m+1)} + \frac{3m^2+11m+4}{2(m+1)}$	$m >= 1;\ n >= 1$
6.	Effectiveness	$m >= 1;\ n >= \frac{\sqrt{(5m^2-4)}+(m-2)}{2}$	No Integer Solution	$m >= 1;\ n >= 1$

The proposed methodology was adopted for the 'Bridge' and 'Abstract Factory' design patterns. 'Bridge' design pattern is a structural GoF pattern, which supports the abstraction from implementation so that they can vary independently [1]. There are two major participants in the Bridge design pattern, whose number may change during system expansion. The participants are Refined Abstraction

and Concrete Implementor. Considering 'n' number of RefinedAbstraction classes and 'm' number of ConcreteImplementor classes in the design pattern solution [30], quality attribute scores and cut-off points are derived for Bridge Design Pattern.

Abstract Factory pattern is a GoF creational pattern, which enables to encapsulate a number of individual factories having a common concern without specifying their concrete classes [1]. 'Abstract Factory' design pattern having two major participants i.e., ConcreteFactory and AbstractProduct classes, which may change in number while expanding the system. Considering 'n' number of AbstractProduct classes and 'm' number of ConcreteFactory classes in the design pattern solution, quality attribute scores and cut-off points are found out for Abstract Factory Design Pattern. The values obtained for the cut-off points for various quality attributes for Visitor, Bridge and Abstract Factory design patterns are provided in Figure **??**.

Chapter 5

Design Pattern Detection

5.1 Introduction

Design patterns are detected by the help of graph isomorphism and normalized cross correlation techniues. Both the system UML class diagram and the design pattern class diagrams are converted into directed graphs. The nodes of the graph act as classes where as the edges connecting the nodes, refer to the relationship among the corresponding classes. Design pttern subgraph is extracted from the system graph using graph isomorphism technique, and with the help of Normalized Cross Correlation (NCC). It is possible to find the percentange existence of the design pattern . We have applied our approach for the detection of five design patterns, such as Composite, Facade, Flyweight, State, and Template Method on four open source software tools.

5.2 Proposed Work

To apply our methodology, some assumptions should be adopted as follows:

Assumption 1 : Graph G, is represented as a 3-tuple entity. G= (V, E, f(E)), where,

1. V = Set of nodes corresponding to classes of a UML class diagram.
2. E is a function: $V \rightarrow V$, corresponds to the set of edges connecting the nodes.
3. $f(E) : (E \rightarrow W_e)$, function relating the edge to a numeric weightage. The value of We depends on the type of relationship among the classes. We have taken certain values to define various relationships which is shown in Table 5.1.

   If any two classes are connected, having more than one relationship, then the relationship weight becomes the multiplication of the individual weights corresponding to the relationship. e.g. for two classes having both Association and Generalization relationship, the edge connecting the classes should have relationship weight = 2*3 = 6. If two classes are not connected by any of these relationships, then the relationship weight becomes 1. Hence W_e = {2,3,5,6,10,15,30}.

Table 5.1: Relationship Weight Table

Sl. No.	**Relationship**	**Relationship Weight**
1.	Association	2
2.	Generalization	3
3.	Realization	5
4.	Other Case or Disconnected	1

Assumption 2: If number of classes in system graph (SG) : S_n and number of classes in pattern graph (PG) : P_n, then

1. SGM corresponds to System Graph Matrix (V, E, f(E)). $SGM[i,j] \in W_e$, 'i' and 'j' are nodes corresponding to classes of system graph.

2. DPM corresponds to Design Pattern Graph Matrix (V, E, f(E)), $DPM[i,j] \in W_e$, 'i' and 'j' are nodes corresponding to classes of design pattern graph.

3. CGM is the Connectivity Graph Matrix (V,E,p); where

$$\begin{aligned} p = 1 \ iff (E) = 1 \\ = 0 \ otherwise \end{aligned} \tag{5.1}$$

   CGM_s is the Connectivity Graph Matrix for System graph (SG) and CGM_d is the connectivity Graph Matrix for Design Pattern Graph (PG).

Assumption 3: Contribution value of the class (CVC): It is the multiplication value of the weights of edges connecting to all of the classes in a class diagram. If 's' and 'd' are the classes in system graph and design pattern graph respectively, then their contribution value of class are :

1. $CVC_s = \prod_{j=1}^{S_n} SGM[s,j] where \ s,j \in SG(V)$, the set of nodes in System Graph.

2. $CVC_d = \prod_{j=1}^{P_n} DPM[d,j] where \ d,j \in PG(V)$, the set of nodes in Design Pattern Graph.

Assumption 4 : We call a class C_s in system grapha(SG) to be the candidate of a class C_d in design pattern graph (PG) if and only if CVC for C_s is a multiple of the CVC for C_d. Candidate Set of a class C_d = S(C_d), in design pattern graph

is the set of all candidate classes C_s in system graph. S(C_d) = C_s where CVC(C_s) is a multiple of CVC(C_d).

Assumption 5: Correspondence Graph (CRG): P_n*S_n graph, where $CRG[i,j]$ = 1 , if class 'j' in system graph is one of the Candidate Set of the class 'i' in pattern graph. i.e.
$CRG[i,j] = 1 if\ \ Cj \in S(Ci), where\ i \in PG(V)\ and\ j \in SG(V)$.

5.2.1 Proposed Methodology

1. Generate xml files corresponding the system class diagram as well as the pattern class diagram.
2. Extract information regarding the relationship among classes in both the graphs corresponding to class diagrams.
3. Find CVC for all the classes in both system graph (SG) and pattern graph (PG).
4. Find candidate set for all the classes of design pattern graph, i.e. S(C_d) is to be found.
5. Use Filtering algorithm to find most probable candidates for all the pattern classes.
6. Find bijective relations between pattern class and the system class and extract the P_n-subgraph, which may contain the pattern instance.
7. Extract P_n-subgraph of Connectivity Graph Matrix for System graph (CGM_s) and compare with the Connectivity Graph Matrix for Design Pattern Graph (CGM_d). If both the matrices are same, then extract the system P_n-subgraph matrix from SGM, denoted as SGM_k , where $k = P_n$.
8. Perform normalized cross correlation (NCC) between the extracted system P_n-subgraph and the pattern graph.
9. From the NCC value, find percentage of matching occurrences.

   The above method is implemented by C++ programming language, which takes the xml files corresponding the system UML diagram and the design pattern diagram and provides total number of fully matched occurrences and partially matched occurrences.

5.2.2 Filtering Algorithm

Algorithm Filtering Candidates

Input : Correspondence Graph (CRG), System Graph Matrix (SGM), Design Patterm Matrix (DPM)

P_n = total number of pattern classes in design pattern graph (PG).

S_n= total number of classes in system graph (SG).

```
for i = 0 to P_n
  for j = 0 to S_n
    value = DPM[i, j];
      for p = 0 to S_n
        if CRG[i, p] = 1
        flag = 0
          for k = 0 to S_n
            if CRG[j, k] = 1 and
            isfactorial(SGM[p, k], value) = true
              flag = 1;
            endif
          endfor
          if flag = 0
            CRG[i, p] = 0
          endif
        endif
      endfor
  endfor
endfor
```

```
Algorithm isfactorial (a, b)
      if a%b = 0
        return true
      else
        return false
      endif
```

5.2.3 Illustration

Figure 5.1 represents the system diagram to be evaluated. Figure 5.2 represents template design pattern diagram. According to the assumptions, the system graph matrix (SGM) and design pattern matrix (DPM) are shown in Figure 5.3 and 5.4 respectively. Similarly, Connectivity Graph Matrix for System graph and Pattern Graph are shown in Figure 5.5 and Figure 5.6 respectively. Contribution value of class (CVC) for all classes in System Diagram and Design Pattern Diagram are as follws:

CVC(SGC1)=1
CVC(SGC2)=1
CVC(SGC3)=3
CVC(SGC4)=5
CVC(SGC5)=3
CVC(SGC6)=3
CVC(SGC7)=5
CVC(SGC8)=2
CVC(PGC1)=1
CVC(PGC2)=1
CVC(PGC3)=3
CVC(PGC4)=3

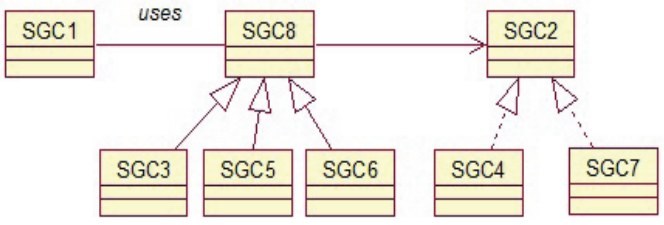

FIGURE 5.1: UML class diagram for System Diagram

The Correspondence Graph is shown in Figure 5.7, in which value CGM[i,j] = 1 indicates the candidateship of system graph class 'j' for pattern class 'i'. After applying filtering algorithm, the updated Correspondence Graph is shown in Figure 5.8.

From the updated Correspondence Graph, bijective mapping are found to further evaluation of the pattern existence in the 4-subgraph extracted from the

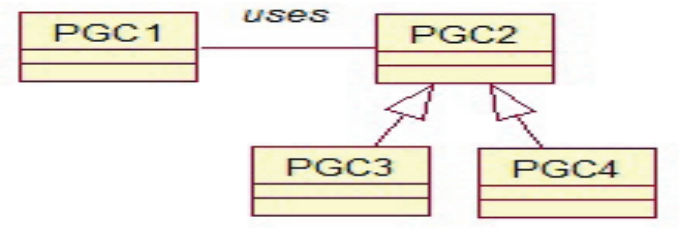

FIGURE 5.2: UML class diagram for Template design pattern

Connectivity Graph Matrix of System Diagram. Let the bijective matching be $(PGC1 \rightarrow SGC1, PGC2 \rightarrow SGC8, PGC3 \rightarrow SGC3, PGC4 \rightarrow SGC5)$. The 4-subgraph of Connectivity Graph Matrix of System Diagram containing SGC1, SGC8, SGC3, SGC5 becomes the same as the Connectivity Graph Matrix of Design Pattern Diagram as shown in Figure 5.9. Hence, the 4-subgraph of System Graph Matrix containing SGC1, SGC8, SGC3, SGC5 as nodes, is extracted from System Graph Matrix, which is shown in Figure 5.9. The normalized cross correlation is applied on the matrix shown in Figure 5.10 with the design pattern graph matrix (DPM), shown in Figure 5.4.

$$NCC = \frac{\sum_{i=1}^{P_n}\sum_{j=1}^{P_n} SGM_4[i,j] * DPM[i,j] - P_n^2 * \mu_s * \mu_d}{\sqrt{(\sum_{i=1}^{P_n}\sum_{j=1}^{P_n} SGM_4[i,j]^2 - P_n^2 * \mu_s^2) * (\sum_{i=1}^{P_n}\sum_{j=1}^{P_n} DPM[i,j]^2 - P_n^2 * \mu_d^2)}} \tag{5.2}$$

where,

$$\mu_s = \frac{1}{P_n^2}\sum_{i=1}^{P_n} SGM_4[i,j] \tag{5.3}$$

$$\mu_d = \frac{1}{P_n^2}\sum_{i=1}^{P_n} DPM4[i,j] \tag{5.4}$$

NCC value becomes 1, which assures 100% or full occurrence of template design pattern in the system graph.

Taking another bijective matching $(PGC1 \rightarrow SGC3, PGC2 \rightarrow SGC8, PGC3 \rightarrow SGC5, PGC4 \rightarrow SGC6)$, the 4-subgraph of Connectivity Graph Matrix of System Diagram containing SGC3, SGC8, SGC5, SGC6 as nodes has been extracted as shown in Figure 5.11 and it does not become the same as the Connectivity Graph Matrix of Design Pattern Diagram as shown in Figure 5.6 . Hence, this set of classes is discarded for further evaluation.

SGC \ SGC	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
3	1	1	1	1	1	1	1	3
4	1	5	1	1	1	1	1	1
5	1	1	1	1	1	1	1	3
6	1	1	1	1	1	1	1	3
7	1	5	1	1	1	1	1	1
8	1	2	1	1	1	1	1	1

FIGURE 5.3: System Graph Matrix (SGM)

PGC \ PGC	1	2	3	4
1	1	1	1	1
2	1	1	1	1
3	1	3	1	1
4	1	3	1	1

FIGURE 5.4: Design Pattern Matrix (DPM)

SGC \ SGC	1	2	3	4	5	6	7	8
1	0	0	0	0	0	0	0	0
2	0	0	0	0	0	0	0	0
3	0	0	0	0	0	0	0	1
4	0	1	0	0	0	0	0	0
5	0	0	0	0	0	0	0	1
6	0	0	0	0	0	0	0	1
7	0	1	0	0	0	0	0	0
8	0	1	0	0	0	0	0	0

FIGURE 5.5: Connectivity Graph Matrix for System Graph(CGM_s)

PGC \ PGC	1	2	3	4
1	0	0	0	0
2	0	0	0	0
3	0	1	0	0
4	0	1	0	0

FIGURE 5.6: Connectivity Graph Matrix for Pattern Graph(CGM_d)

PGC \ SGC	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	1
2	1	1	1	1	1	1	1	1
3	0	0	1	0	1	1	0	0
4	0	0	1	0	1	1	0	0

FIGURE 5.7: Correspondence Graph(CRG)

PGC \ SGC	1	2	3	4	5	6	7	8
1	1	1	1	1	1	1	1	0
2	0	0	0	0	0	0	0	1
3	0	0	1	0	1	1	0	0
4	0	0	1	0	1	1	0	0

FIGURE 5.8: Updated Correspondence Graph(CRG) after Applying Candidate Filtering Algorithm

SGC \ SGC	1	8	3	5
1	0	0	0	0
8	0	0	0	0
3	0	1	0	0
5	0	1	0	0

FIGURE 5.9: 4-subgraph of CRG (CRG_4) - Testcase1

SGC \ SGC	1	8	3	5
1	1	1	1	1
8	1	1	1	1
3	1	3	1	1
5	1	3	1	1

FIGURE 5.10: 4-subgraph of SGM (SGM_4) - Testcase1

SGC \ SGC	3	8	5	6
3	0	1	0	0
8	0	0	0	0
5	0	1	0	0
6	0	1	0	0

FIGURE 5.11: 4-subgraph of CRG (CRG_4) - Testcase2

Sl. No.	Design Pattern	Jrat 100%	*Jrat* $\geq$ 90%	Junit 100%	*Junit* $\geq$ 90%	Lexi-Alpha 100%	*Lexi – Alpha* $\geq$ 90%	Informa 100%	*Informa* $\geq$ 90%
1.	Composite	71	0	0	0	182	0	107	0
2.	Facade	124	705	564	1608	0	0	0	0
3.	Flyweight	600	8409	707	5439	581	1595	201	4759
4.	State	124	14	564	492	0	0	0	0
5.	Template Method	286	4	6098	977	522	6	4422	866

FIGURE 5.12: Implementation Results

5.3 Implementation and Results

The proposed approach is applied on four widely adopted open souce softwares having toal no. of classes ranging from 76 to 109. We have performed our method on Jrat, Junit, Lexi-alpha and Informa tools for evaluation of the existence of 5 design patterns, such as Composite, Facade, Flyweight, State and Template Method design patterns. Results for 100% existence and partial existence more than 90% are shown in Figure 5.12.

Chapter 6

Conclusion and Future Work

6.1 Formalization of Design Pattern

Design patterns are based on UML diagrams, which support semi-formal notation to design a particular system. In this study, an attempt has been made to propose a grammar which satisfies the design pattern language and formally verifies the security patterns. In order to demonstrate this approach, a case study on online banking system has been considered. For this case study, extended UML class diagram visualizing design patterns is generated by using UML class diagram. *Single Access Point*, provides a single login screen to all external entities of the system, which helps the system to trace the unusual requests thus maintaining the *availability* of the system for other entities. *Check Point* ensures the *confidentiality* of system by authenticating the user and it also enforce certain security policies and penalizes the user for violating security policies. The *role-based access control (RBAC)* maintains the integrity of the system authorizing the user with the help of user-role-privilege relationship. RBAC also improves the *confidentiality* of the system by providing access rights.

In future, a good number of design patterns can be added in order to extend the pattern language. As per now, the association of classes is being checked by the tool. Grammar can be extended for verifying the operations of class and role played by these operations under design pattern. Prposed approach can be encorporated as a plugin or an extension for widely used UML drawing software solutions such as IBM Rational Rose. Basic fundamental advantage of security patterns is reusability, for this purpose, XML file can be generated and saved for further use, also skeleton source code can be generated out of the UML class diagram for several programming languages.

6.2 Quantitative Analysis of Quality of Design Patterns

We have suggested a methodology to assess the effects of design patterns in an object-oriented system environment. Using QMOOD approach, object-oriented metrics are calculated in terms of the number of classes and their relationships assumed in a UML class diagram. The cut-off points are calculated in order to provide the exact size of the system in terms of the number of classes, for

which design pattern solution provides better result in terms of quality attribute as compared to the non-pattern solution for the same system. This work can fortify the goal-oriented design making, since it is expected that every design attribute demands a categorical solution, according to its special needs with reference to quality.

Also, the suggested approach to assess the effects of design patterns in an object-oriented system environment, can be applicable for other design patterns and similar estimation can be done in order to find out various quality improver. This methodology can be extended to compare the quality of system which uses multiple number of design patterns.

6.3 Design Pattern Detection

This method provides a novel way to detect design patterns from the source code of a software. In the field of Software Reverse Engineering, this approach to detect design pattern instances in a software, is quite adoptable as it automatically detects design patterns. The use of Normalized Cross Correlation method in the process of design pattern detection provides a way not only to detect full occurrence of the pattern but it also provides a measure to find the percentage matching of the pattern. This method is useful for software engineers to get knowledge about the pattern existence in the system.

As a future work, this approach can be applicable for other design patterns and similar evaluation can be done for various open source softwares. The proposed approach can be adopted to develop a plugin or an extension for Eclipse IDE which will take the source codes of the projects as input and produce the total number of matching occurrences of design patterns as output.

Bibliography

[1] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[2] Deepak Alur, Dan Malks, John Crupi, Grady Booch, and Martin Fowler. *Core J2EE Patterns (Core Design Series): Best Practices and Design Strategies*. Prentice Hall, 2nd edition, 2003.

[3] Martin Fowler. *Patterns of enterprise application architecture*. Addison-Wesley, Boston, USA, 2002.

[4] Markus Schumacher, Eduardo Fernandez-Buglioni, Duane Hybertson, Frank Buschmann, and Peter Sommerlad. *Security Patterns: Integrating security and systems engineering*. John Wiley & Sons, West Sussex, England, 2005.

[5] Christopher Steel, Ramesh Nagappan, and Ray Lai. *Core Security Patterns: Best Practices and Strategies for J2EE, Web Services, and Identity Management*. Prentice Hall PTR, 2005.

[6] Ashish Kumar Dwivedi and Santanu Kumar Rath. Incorporating security features in service-oriented architecture using security patterns. *ACM SIGSOFT Software Engineering Notes*, 40(1):1–6, 2015.

[7] Jörg Niere, Wilhelm Schäfer, Jörg P Wadsack, Lothar Wendehals, and Jim Welsh. Towards pattern-based design recovery. In *Proceedings of the 24th international conference on Software engineering*, pages 338–348. ACM, 2002.

[8] Hong Zhu and Ian Bayley. An algebra of design patterns. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 22(3):23:1–23:35, July 2013.

[9] Joseph Yoder and Jeffrey Barcalow. Architectural patterns for enabling application security. In *In proceeding of the 4th Conference on Patterns Language of Programming (PLoP'97)*, 1997.

[10] Robert Hanmer. *Patterns for fault tolerant software.* John Wiley & Sons, 2007.

[11] Jing Dong and Sheng Yang. Extending uml to visualize design patterns in class diagrams. In *Proceedings of the Fifteenth International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 124–131. San Francisco Bay, California, USA, 2003.

[12] F Brito e Abreu. The mood metrics set. In *Proceedings of the 9th European Conference on Object-Oriented Programming (ECOOP 95) Workshop Metrics*, volume 95, 1995.

[13] Shyam R Chidamber, David P Darcy, and Chris F Kemerer. Managerial use of metrics for object-oriented software: An exploratory analysis. *Software Engineering, IEEE Transactions on*, 24(8):629–639, 1998.

[14] Jagdish Bansiya and Carl G. Davis. A hierarchical model for object-oriented design quality assessment. *Software Engineering, IEEE Transactions on*, 28(1):4–17, 2002.

[15] Hausi A Müller, Jens H Jahnke, Dennis B Smith, Margaret-Anne Storey, Scott R Tilley, and Kenny Wong. Reverse engineering: A roadmap. In *Proceedings of the Conference on the Future of Software Engineering*, pages 47–60. ACM, 2000.

[16] Jean-Marie Favre. Cacophony: Metamodel-driven software architecture reconstruction. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 204–213. IEEE, 2004.

[17] Rudolf Ferenc, Arpad Beszedes, Lajos Fulop, and Janos Lele. Design pattern mining enhanced by machine learning. In *Software Maintenance, 2005. ICSM'05. Proceedings of the 21st IEEE International Conference on*, pages 295–304. IEEE, 2005.

[18] Yann-Gaël Guéhéneuc, Houari Sahraoui, and Farouk Zaidi. Fingerprinting design patterns. In *Reverse Engineering, 2004. Proceedings. 11th Working Conference on*, pages 172–181. IEEE, 2004.

[19] Heyuan Huang, Shensheng Zhang, Jian Cao, and Yonghong Duan. A practical pattern recovery approach based on both structural and behavioral analysis. *Journal of Systems and Software*, 75(1):69–87, 2005.

[20] Eclipse Plugin-ObjectAid. http://www.objectaid.com/.

[21] John Vlissides. Notation, Notation, Notation. C++ Report. Technical report, April 1998.

[22] Jing Dong. Uml extensions for design pattern compositions. *Journal of object technology*, 1(5):151–163, 2002.

[23] Stefan Berner, Martin Glinz, and Stefan Joos. A classification of stereotypes for object-oriented modeling languages. In *Proceedings of the Second International Conference on the Unified Modeling Language (UML), LNCS1723,*, pages 249–264. Springer-Verlag, October 1999.

[24] Toufik Taibi and David Chek Ling Ngo. Formal specification of design patterns - a balanced approach. *Journal of Object Technology*, 2(4):127–140, 2003.

[25] Jing Dong, Tu Peng, and Yajing Zhao. Automated verification of security pattern compositions. *Information and Software Technology*, 52(3):274–295, 2010.

[26] Ashish Kumar Dwivedi and Santanu Kumar Rath. Analysis of a complex architectural style c2 using modeling language alloy. *Information and Software Technology*, 3(2):152–164, 2014.

[27] Ian Bayley and Hong Zhu. Formal specification of the variants and behavioral features of design patterns. *Journal of Systems and Software*, 83(2):209–221, 2010.

[28] Ashish Kumar Dwivedi and Santanu Kumar Rath. Selecting and formalizing an architectural style: A comparative study. In *Contemporary Computing (IC3), 2014 Seventh International Conference on*, pages 364–369. IEEE, 2014.

[29] Shouvik Dey and Swapan Bhattacharya. Formal specification of structural and behavioral aspects of design patterns. *Journal of Object Technology*, 9(6):99–126, 2010.

[30] Apostolos Ampatzoglou, Georgia Frantzeskou, and Ioannis Stamelos. A methodology to assess the impact of design patterns on software quality. *Information and Software Technology*, 54(4):331–346, 2012.

[31] Imène Issaoui, Nadia Bouassida, and Hanêne Ben-Abdallah. Using metric-based filtering to improve design pattern detection approaches. *Innovations in Systems and Software Engineering, Springer*, December 2014.

[32] Chih-Hung Chang, Chih-Wei Lu, and Pao-Ann Hsiung. Pattern-based framework for modularized software development and evolution robustness. *Information and Software Technology*, 53(4):307–316, 2011.

[33] Nien-Lin Hsueh, Peng-Hua Chu, and William Chu. A quantitative approach for evaluating the quality of design patterns. *Journal of Systems and Software*, 81(8):1430–1439, 2008.

[34] Apostolos Ampatzoglou, Sofia Charalampidou, and Ioannis Stamelos. Research state of the art on gof design patterns: A mapping study. *Journal of Systems and Software*, 86(7):1945–1964, 2013.

[35] Jing Dong, Yang Sheng, and Kang Zhang. Visualizing design patterns in their applications and compositions. *Software Engineering, IEEE Transactions on*, 33(7):433–453, 2007.

[36] Yoshio Kataoka, Takeo Imai, Hiroki Andou, and Tetsuji Fukaya. A quantitative evaluation of maintainability enhancement by refactoring. In *Software Maintenance, 2002. Proceedings. International Conference on*, pages 576–585. IEEE, 2002.

[37] Brian Huston. The effects of design pattern application on metric scores. *Journal of Systems and Software*, 58(3):261–269, 2001.

[38] Hervé Albin-Amiot and Yann-Gaël Guéhéneuc. Meta-modeling design patterns: Application to pattern detection and code synthesis. In *Proceedings of ECOOP Workshop on Automating Object-Oriented Software Development Methods*, 2001.

[39] Dirk Heuzeroth, Thomas Holl, Gustav Hogstrom, and Welf Lowe. Automatic design pattern detection. In *Program Comprehension, 2003. 11th IEEE International Workshop on*, pages 94–103. IEEE, 2003.

[40] Federico Bergenti and Agostino Poggi. Improving uml designs using automatic design pattern detection. In *12th International Conference on Software Engineering and Knowledge Engineering (SEKE)*, pages 336–343. Citeseer, 2000.

[41] Manjari Gupta, R Singh Rao, and Anil Kumar Tripathi. Design pattern detection using inexact graph matching. In *Communication and Computational Intelligence (INCOCCI), 2010 International Conference on*, pages 211–217. IEEE, 2010.

[42] Sven Wenzel and Udo Kelter. Model-driven design pattern detection using difference calculation. In *1st Int. Workshop on Pattern Detection for Reverse Engineering, Co-located with 13th Working Conf. on Reverse Engineering*, 2006.

[43] Giuliano Antoniol, Gerardo Casazza, Massimiliano Di Penta, and Roberto Fiutem. Object-oriented design patterns recovery. *Journal of Systems and Software*, 59(2):181–196, 2001.

[44] Manjari Gupta, Akshara Pande, R Singh Rao, and AK Tripathi. Design pattern detection by normalized cross correlation. In *Methods and Models in Computer Science (ICM2CS), 2010 International Conference on*, pages 81–84. IEEE, 2010.

[45] Afnan Salem Ba-Brahem and M Qureshi. The proposal of improved inexact isomorphic graph algorithm to detect design patterns. *arXiv preprint arXiv:1408.6147*, 2014.

[46] Nikolaos Tsantalis, Alexander Chatzigeorgiou, George Stephanides, and Spyros T Halkidis. Design pattern detection using similarity scoring. *Software Engineering, IEEE Transactions on*, 32(11):896–909, 2006.

[47] Jing Dong, Yongtao Sun, and Yajing Zhao. Design pattern detection by template matching. In *Proceedings of the 2008 ACM symposium on Applied computing*, pages 765–769. ACM, 2008.

[48] Yu Dongjin, Jianlin Ge, and Wei Wu. Detection of design pattern instances based on graph isomorphism. In *Software Engineering and Service Science (ICSESS), 2013 4th IEEE International Conference on*, pages 874–877. IEEE, 2013.